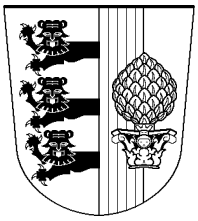
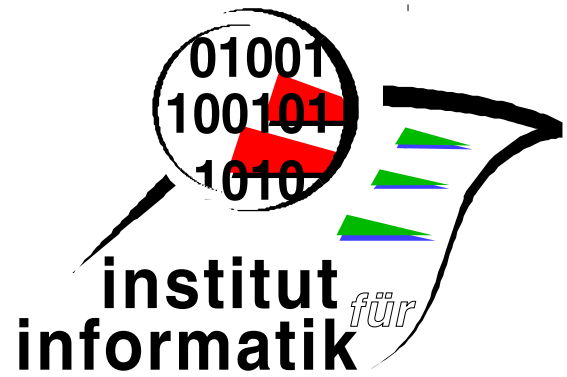
Universität Augsburg



Linked Lists Calculated

Bernhard Möller

Report 1997-07 Dezember 1997



Institut für Informatik

D-86135 Augsburg

Linked Lists Calculated

Bernhard Möller

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany. email: moeller@uni-augsburg.de

Abstract. We use a relational calculus of pointer structures to calculate a number of standard algorithms on singly linked lists, both acyclic and cyclic. This shows that our techniques are not just useful for tree-like structures, but apply to general pointer structures as well.

1 Introduction

Although pointer algorithms are very error-prone they lie at the very heart of many implementations. Yet they have received surprisingly little attention in work on formal derivation and verification of programs. If they are treated, mostly formulas from predicate logic are used, which tend, however, to be very complex and unwieldy. A more algebraic approach was presented in the work of Berger et al. (1991) and Möller (1991–1993) and developed into more general form by Möller (1997). However, in the latter paper only examples with tree-like structures were treated. We show that the approach covers cyclic structures as well. In fact, we demonstrate that the derivations for the acyclic case can be carried over to the cyclic case quite simply and hence be re-used. Proofs that are missing from the present paper are straightforward or can be found in Möller (1997).

2 Relational Notation

Our prominent mathematical tool are binary relations by which we model the directed graph underlying a pointer structure and describe accessibility and sharing. Given a set X we denote its power set by $\wp(X)$. Now the set of all *binary relations* between sets M and N is $M \leftrightarrow N \stackrel{\text{def}}{=} \wp(M \times N)$. We use the notations $R \in M \leftrightarrow N$ and $R : M \leftrightarrow N$ synonymously. By $\mathsf{dom}R$ and $\mathsf{ran}R$ we denote domain and range of relation R. The *converse* $R^\smile : N \leftrightarrow M$ of R is given by $R^\smile \stackrel{\text{def}}{=} \{(y,x) : (x,y) \in R\}$. The *image* of set $L \subseteq M$ under R is $R(L) \stackrel{\text{def}}{=} \{y : \exists\, x \in L : (x,y) \in R\}$.

Particularly for analyzing the reachable part of a pointer structure we shall use the *domain restriction* of R to a subset $L \subseteq M$ given by $L \lhd R \stackrel{\text{def}}{=} R \cap L \times N$. Dually, the *range restriction* of R to a subset $L \subseteq N$ is $R \rhd L \stackrel{\text{def}}{=} R \cap M \times L$.

The *composition* $R\,;S : M \leftrightarrow P$ of two relations $R : M \leftrightarrow N$ and $S : N \leftrightarrow P$ is defined as $R\,;S \stackrel{\text{def}}{=} \{(x,z) : \exists\, y \in N : (x,y) \in R \wedge (y,z) \in S\}$. Left and right neutral elements for R w.r.t. this operation are provided by I_M and I_N, where for a set P one defines the *identity relation* $I_P : P \leftrightarrow P$ by $I_P \stackrel{\text{def}}{=} \{(x,x) : x \in P\}$. The index $_P$ will be omitted when P is clear from the context. As usual, R^+ and R^* are the transitive and the reflexive-transitive closures of relation R.

Relation $R \subseteq M \times N$ is called a *(partial) map* if $R^{\smile}; R \subseteq I_N$. We write $R : M \rightsquigarrow N$ to indicate that R is a map. For further notions and laws concerning relations consider e.g. Schmidt, Ströhlein (1993).

3 Stores and Pointer Structures

A pointer structure consists of a set of records connected by pointers. Let $\mathcal{A}$ be a set of *records* (represented, say, by their initial addresses). We assume a distinguished element $\diamond \in \mathcal{A}$ which plays the role of nil in Pascal or NULL in C, i.e., serves as a terminal pseudo-node for the underlying graph. The elements of $\mathcal{A} \backslash \{\diamond\}$ are called *proper* records. Let, moreover, $(\mathcal{N}_j)_{j \in J}$ be a family of sets of *node values*, such as integers or Booleans.

Then a *record scheme* consists of a non-empty set K of *selectors* each with a type $\mathcal{A} \to \mathcal{A}$ or $\mathcal{A} \to \mathcal{N}_j$ for some $j \in J$. Given such a record scheme, a *store* is a family $S = (S_k)_{k \in K}$ of partial maps such that

1. $S_k : \mathcal{A} \rightsquigarrow \mathcal{A}$ if k has type $\mathcal{A} \to \mathcal{A}$,
2. $S_k : \mathcal{A} \rightsquigarrow \mathcal{N}_j$ if k has type $\mathcal{A} \to \mathcal{N}_j$ and
3. $\diamond \notin \mathit{recs}(S) \stackrel{\text{def}}{=} \bigcup_{k \in K} \mathsf{dom} S_k$, the set of records allocated in S.

A store may be viewed as a labeled directed graph: the selectors are the arc labels and S_k is the set of arcs labeled by k. We record these sets separately to be able to model updating along a single selector adequately. The requirement that the S_k be maps reflects the uniqueness of selection in records. By the third requirement, in a store, $\diamond$ is not related to anything and hence cannot be "dereferenced". The relational operations are extended componentwise to stores.

Our running example of a record scheme is one for singly linked lists. Assume a set $\mathcal{N}$ of node values and two selectors $\mathit{head}, \mathit{tail}$ of types $\mathit{head} : \mathcal{A} \to \mathcal{N}$ and $\mathit{tail} : \mathcal{A} \to \mathcal{A}$. Then a list store L consists of two partial maps $L_{\mathit{head}} : \mathcal{A} \rightsquigarrow \mathcal{N}$ and $L_{\mathit{tail}} : \mathcal{A} \rightsquigarrow \mathcal{A}$, where L_{head} returns the node value and L_{tail} gives the next record in the list.

Frequently we want to abstract from the node values of the records and consider just their interrelationship through the pointers. For a store $S = (S_k)_{k \in K}$, this is modeled by the binary *access relation* $[S] \subseteq \mathcal{A} \times \mathcal{A}$ given by

$$[S] \stackrel{\text{def}}{=} \bigcup_{k \in J} S_k \ ,$$

where $J \subseteq K$ is the set of all selectors k of type $\mathcal{A} \to \mathcal{A}$. In the graph view, this operation "forgets" the arc labels. For instance, the access relation for a list store L is $[L] \stackrel{\text{def}}{=} L_{\mathit{tail}}$.

Let now P denote the set of all stores for a given record scheme. The set of *entries* to pointer structures is $\mathcal{A}^+$, the set of all non-empty finite lists of elements of $\mathcal{A}$. We choose lists rather than sets or bags of entries, since in pointer algorithms both order and multiplicity of entries may be relevant.

Now a *pointer structure* is an element of $\mathcal{P} \stackrel{\text{def}}{=} A^+ \times P$. For convenience we introduce the functions

$$ptr(s,S) \stackrel{\text{def}}{=} s\ , \qquad sto(s,S) \stackrel{\text{def}}{=} S\ , \qquad recs(s,S) \stackrel{\text{def}}{=} recs(S)\ .$$

In denoting lists of entries we separate the elements by commas. So a pointer structure will be written in the form $x_1, \ldots, x_n, S$ with entries x_i and store S.

4 Reachability and Sharing

In a pointer structure $(s,S) \in \mathcal{P}$ we can follow the pointers from the entries s to other records. This is modeled by the function $reach : \mathcal{P} \to \wp(A)$ with

$$reach(s,S) \stackrel{\text{def}}{=} [S]^*(\mathsf{set}\, s)\ .$$

Here $\mathsf{set}\, s$ is the set of elements occurring in $s \in \mathcal{A}^+$. From this definition it is straightforward that

$$reach(x_1, \ldots, x_n, S) = reach(x_1, S) \cup \cdots \cup reach(x_n, S)\ . \tag{1}$$

Associated with *reach* is the *reachability relation* $\vdash : \mathcal{P} \leftrightarrow \wp(A^+)$ given by

$$p \vdash L \stackrel{\text{def}}{\Leftrightarrow} reach(p) \cap \mathsf{set}\, L \neq \emptyset\ ,$$

where $\mathsf{set}\, L = \bigcup_{s \in L} \mathsf{set}\, s$. So this relation holds iff some record in the entries in L is accessible from the entries of p. For singleton set L we will omit the set braces.

Moreover, we introduce a unary predicate *sharing* on $\mathcal{P}$ by setting

$$sharing(n_1, \ldots, n_k, S) \stackrel{\text{def}}{\Leftrightarrow} \bigvee_{i=1}^{k} \bigvee_{j=i+1}^{k} reach(n_i, S) \cap reach(n_j, S) \not\subseteq \{\diamond\}\ .$$

So a pointer structure shows sharing iff a proper record is reachable from two of its entries. Note that this predicate is independent of the order of the entries n_i but not of their multiplicity. So if a record occurs twice in a list of entries, there will be sharing, as expected.

The reachable set abstracts too much from the actual contents of the store in a pointer structure. Therefore we characterize additionally that part of store S that is reachable from s by the restriction

$$from(s,S) \stackrel{\text{def}}{=} (s,\ reach(s,S) \bowtie S)\ ,$$

i.e., the substructure in which only the contents of records reachable from the entries s are kept. The restriction is again taken componentwise, i.e., for all $k \in K$.

5 Pointer Implementations

5.1 Reasonable Abstraction Functions

We now consider implementations of abstract objects of some set $\mathcal{O}$ by pointer structures in such a way that each object is represented by a pointer structure $(n, S) \in \mathcal{P}$ with a single entry $n \in \mathcal{A}$. As usual (see e.g. Hoare (1972)), the relation between abstract and concrete levels is established by a partial abstraction function $F : \mathcal{A} \times P \rightsquigarrow \mathcal{O}$ such that F is surjective. To allow representations of *tuples* of abstract objects, we extend F to a partial function $F : \mathcal{P} \rightsquigarrow \mathcal{O}^+$ on arbitrary pointer structures by setting $F(n_1 \cdots n_k, S) \stackrel{\text{def}}{=} F(n_1, S) \cdots F(n_k, S)$. As usual, F induces an equivalence relation $\sim$ on $\mathcal{P}$ by

$$p \sim q \stackrel{\text{def}}{\Leftrightarrow} F(p) = F(q) \ .$$

Since the pointer representation of an abstract object should be essentially determined by the entries to the structure, we say that an abstraction function is *reasonable* if for all $p, q \in \mathcal{P}$ we have

$$\mathit{from}(p) = \mathit{from}(q) \Rightarrow p \sim q \ .$$

This seemingly simple concept is the key idea that makes our treatment work uniformly and independently of particular data structures such as lists or trees. It allows us to reduce questions about the changes a selective updating effects to a much simpler analysis of the changes in reachability. In particular, we can use the well-established relational calculus for that analysis.

5.2 Implementation of Operations

As usual (see e.g. Hoare (1972)), the general pattern for transferring operations from abstract level to pointer level is as follows.

Consider an operation of type $\mathcal{O}^n \rightsquigarrow \mathcal{B}$ that leads into a set $\mathcal{B}$ of "external" values such as integers or Booleans. We define an *implementation relation* $OPOI \in (\mathcal{P} \rightsquigarrow \mathcal{B}) \leftrightarrow (\mathcal{O}^n \rightsquigarrow \mathcal{B})$ by setting

$$pg \ OPOI \ g \stackrel{\text{def}}{\Leftrightarrow} pg = F \,;\, g \ .$$

So the implementation pg has to mimic the specification g faithfully. Note the implicit use of the extended abstraction function F for the representation of tuples in $\mathcal{O}^n$.

For operations of type $\mathcal{O}^n \rightsquigarrow \mathcal{O}$ we are more liberal and allow the implementation to be non-deterministic, i.e., a relation rather than a map. This is reasonable, since different concrete objects may represent the same abstract object. A typical non-deterministic operation at the pointer level would be the allocation of new records (see Möller (1997)). Our notion of implementation will be parameterized by additional requirements on the implementing relation, such as preservation of certain aspects of the store. Such requirements are again formulated as relations between "old" and "new" pointer structures. Hence our *implementation relation* has type $POI : (\mathcal{P} \leftrightarrow \mathcal{P}) \to ((\mathcal{P} \leftrightarrow \mathcal{P}) \leftrightarrow (\mathcal{O}^n \rightsquigarrow \mathcal{O}))$ and is defined by

$$pf \ POI(req) \ f \stackrel{\text{def}}{\Leftrightarrow} pf \,;\, F = F \,;\, f \wedge pf \subseteq req \ .$$

Here *req* is the additional requirement, examples of which will be given later.

The most liberal specification $POI(ALL)$, where $ALL \stackrel{\text{def}}{=} \mathcal{P} \times \mathcal{P}$ is the universal relation, does not exclude indirect side-effects on parts of p that point into the reachable part $from(p)$. We also want to give stronger specifications that guarantee that changes take place only in the relevant reachable part or outside the current store, i.e., on "new" records. To this end we define the set

$$noreach(p) \stackrel{\text{def}}{=} recs(p) \backslash reach(p) = recs(p) \backslash recs(from(p)) \ .$$

It is the set of all records that are not reachable from the entries of p and hence should better be left alone by changes to the store of p. Note that

$$n \in noreach(p) \Leftrightarrow n \in recs(p) \wedge p \not\vdash n \ . \tag{2}$$

Now we can define two constraining relations $loc, pres \in \mathcal{P} \leftrightarrow \mathcal{P}$ by setting

$$\begin{aligned} p \ loc \ q &\stackrel{\text{def}}{\Leftrightarrow} noreach(p) \bowtie sto(p) = noreach(p) \bowtie sto(q) \ , \\ p \ pres \ q &\stackrel{\text{def}}{\Leftrightarrow} noreach(p) \subseteq noreach(q) \ . \end{aligned}$$

So *loc* requires that the part of the store that is unreachable in p is left untouched in q; by our definition this does, however, not exclude adding new records to the store.

However, *loc* also holds if in the "modified" structure q there are pointers into $noreach(p)$. So records that were unreachable in p may become reachable by the modification and hence accessible for subsequent modification. This potential source of problems for updates in q through $ptr(p)$ is excluded by postulating *pres*.

Now we can work with the strengthenings $POI(loc)$, $POI(pres)$ or even $POI(loc \cap pres)$ of $POI(ALL)$. They all still admit implementation by copying and by re-use.

5.3 Development Strategy

To calculate a pointer implementation pf of $f : \mathcal{O}^n \rightsquigarrow \mathcal{O}$, we start with the expression $f(F(p))$ and try to transform it by equational reasoning into an expression $F(E)$ such that $F(E) = f(F(p))$ and E does not contain F. Then we can define pf by setting $pf(p) \stackrel{\text{def}}{=} E$ and are sure that $pf \ POI(ALL) \ f$ holds. Design decisions are reflected by the particular choice of the applied equations and generally result in a reduction of nondeterminacy. For implementations of operations $g : \mathcal{O} \rightsquigarrow \mathcal{N}$ we may, more directly, start with the expression $g(F(p))$ and transform it in such a way that F is eliminated from it.

One design goal is to keep changes to a minimum. This has two aspects:

– preserve the entries to pointer structures, if possible;

– implement changes to single record components by selective updating, if possible.

We shall see these goals influence our example derivations. In particular, they will motivate the introduction of strengthened requirements as additional invariants.

6 Overwriting Pointer Structures

To describe selective updating, we use the operation of *overwriting*. Given relations $R, S : M \leftrightarrow N$, the relation $R \mid S : M \leftrightarrow N$ (pronounced "R onto S") is given by

$$R \mid S \stackrel{\text{def}}{=} R \cup \overline{\mathsf{dom} R} \lhd S ,$$

where $\overline{\mathsf{dom} R}$ is the complement of $\mathsf{dom} R$. Hence $R \mid S$ results from S by changing the image sets according to the prescription of R (if any). For example, if S is a map then $\{(x, y)\} \mid S$ "updates" S to make y the value corresponding to x. The set $M \leftrightarrow N$ forms a monoid under $\mid$ with $\emptyset$ as its neutral element. Moreover, the set of $M \rightsquigarrow N$ of maps is a submonoid of $M \leftrightarrow N$. For further properties see Möller (1993b).

Consider now two stores S and T over the same record scheme. The *overwriting* $T \mid S$ is again defined componentwise. For store S and pointer structure q we set

$$S \mid q \stackrel{\text{def}}{=} (ptr(q), S \mid sto(q)) .$$

Lemma 6.1 $p \not\in recs(S) \Rightarrow reach(S \mid p) = reach(p)$.

In selective updating only one of the component maps of a store is overwritten properly. A store which models the update along selector k is $(x \stackrel{k}{\mapsto} y)$ given by

$$\begin{aligned} (x \stackrel{k}{\mapsto} y)_k &\stackrel{\text{def}}{=} \{(x, y)\} , \\ (x \stackrel{k}{\mapsto} y)_j &\stackrel{\text{def}}{=} \emptyset \qquad\qquad \text{for } j \neq k . \end{aligned}$$

To ease the notation and to keep with traditional programming languages, we introduce an operation $\_.\_ := \_ : \mathcal{P} \times K \times \mathcal{P} \to \mathcal{P}$ for selective updating:

$$\begin{aligned} (n, S).k := (m, T) &\stackrel{\text{def}}{=} (n, (n \stackrel{k}{\mapsto} m) \mid T) \quad \text{if } k \text{ has type } \mathcal{A} \to \mathcal{A}, \\ (n, S).k := x &\stackrel{\text{def}}{=} (n, (n \stackrel{k}{\mapsto} x) \mid S) \quad \text{if } k \text{ has type } \mathcal{A} \to \mathcal{N}_j. \end{aligned}$$

Moreover, we define the selection operation $\_.\_ : \mathcal{P} \times K \rightsquigarrow (\mathcal{A} \cup \bigcup_{j \in K} \mathcal{N}_j)$ by

$$\begin{aligned} (n, S).k &\stackrel{\text{def}}{=} (S_k(n), S) \text{ if } k \text{ has type } \mathcal{A} \to \mathcal{A} \text{ and } S_k(n) \text{ is defined}, \\ (n, S).k &\stackrel{\text{def}}{=} S_k(n) \qquad \text{if } k \text{ has type } \mathcal{A} \to \mathcal{N}_j. \end{aligned}$$

Otherwise, $(n, S).k$ is undefined. When using selections as arguments, undefinedness is assumed to propagate, according to the strictness of relational semantics. We have the following properties:

Lemma 6.2 Let $n \stackrel{\text{def}}{=} ptr(p)$ and $r \stackrel{\text{def}}{=} (n, sto(q))$.

1. $ptr(p.k := q) = ptr(p)$.
2. $(p.k := q).k = (ptr(p) \stackrel{k}{\mapsto} ptr(q)) \mid q$.
3. $j \neq k \Rightarrow (p.k := q).j = (ptr(p) \stackrel{k}{\mapsto} ptr(q)) \mid r$ where $r \stackrel{\text{def}}{=} (ptr(p), sto(q))$.
4. $(p.k := p.k) = p$.

5. $j : \mathcal{A} \to \mathcal{N}_i \wedge j \neq k \Rightarrow (p.k := q).j = p.j$.
6. $q \not\models n \wedge k : \mathcal{A} \to \mathcal{A} \Rightarrow \mathit{from}((p.k := q).k) = \mathit{from}(q)$.
7. $j : \mathcal{A} \to \mathcal{A} \wedge j \neq k \wedge r.j \not\models n \Rightarrow \mathit{from}((p.k := q).j) = \mathit{from}(r.j)$.
8. $q \not\models L \Rightarrow q \notin \mathsf{set}\, L \wedge q.k \not\models L$.
9. $\neg\ \mathit{sharing}(m, n, S) \Rightarrow \neg\ \mathit{sharing}(S_k(m), n, S)$.
10. $\mathit{noreach}(p) \subseteq \mathit{noreach}(p.j)$, i.e., $p\ \mathit{pres}\ p.j$.
11. $\mathit{noreach}((n, S).k := (m, T)) = \mathit{noreach}(m, \overline{\{n\}} \bowtie T)$.

For pointer implementations that use selective updating it usually is important that the updates work locally. This can be established using the following localization property:

Lemma 6.3 Assume that abstraction function F is reasonable and $k : \mathcal{A} \to \mathcal{A}$.

1. $q \not\models \mathit{recs}(S) \Rightarrow S \mid q \sim q$.
2. $q \not\models \mathit{ptr}(p) \Rightarrow (p.k := q).k \sim q$.
3. Let $r \stackrel{\text{def}}{=} (\mathit{ptr}(p), \mathit{sto}(q))$. Then $j \neq k \wedge r.j \not\models \mathit{ptr}(p) \Rightarrow (p.k := q).j \sim r.j$.

7 Acyclic Stores and Forests

We have seen that many properties depend on the absence of sharing. This is guaranteed by forests, which are therefore of special interest. For their characterization we need two notions about binary relations. A relation $R : M \leftrightarrow N$ is *acyclic* iff $R^+ \cap I = \emptyset$. Hence R is acyclic iff no element is reachable from itself via a non-empty path. R is *injective* iff $R\,;R^{\smile} \subseteq I$, i.e., iff no two distinct elements have a common successor under R. If $T \subseteq S$ and S is acyclic or injective, then so is T, since all operations involved in the characterizations of these notions are monotonic w.r.t. inclusion.

These notions are carried over to stores as follows. A store S is called *acyclic* if $[S]$ is acyclic, and *injective* if $[S] \bowtie \overline{\{\diamond\}}$ is injective. This means that no two different records point to the same *proper* record or, equivalently, that the underlying directed graph has maximal in-degree 1, except perhaps at the pseudo-record $\diamond$. Finally, S is called a *forest* if it is acyclic and injective. We have the following separation properties for acyclic stores (and hence forests) which will allow localization of side effects:

Lemma 7.1 1. Let S be injective. Then for all $x, y \in \mathcal{A}$ we have

$$\mathit{sharing}(x, y, S) \Rightarrow ((y, S) \vdash x \vee (x, S) \vdash y)\ .$$

2. Let S be acyclic and assume $y \in [S]^+(x)$. Then $(y, S) \not\models x$.
3. Let S be acyclic and assume $y \in [S]^+(x)$. Then $\forall\ z \in \mathcal{A} : \neg\ \mathit{sharing}(z, x, S) \Rightarrow \neg\ \mathit{sharing}(z, y, S)$.
4. Let S be a forest and y, z two distinct successors of x under $[S]$, i.e., assume $y, z \in [S](x) \wedge y \neq z$. Then $\neg\ \mathit{sharing}(x, y, S)$.
5. Let S be a forest and assume $y \in [S](x)$. Then

$$\mathit{noreach}(y, S) = \mathit{noreach}(x, S) \cup \{x\} \cup \bigcup_{z \in [S](x) \setminus \{y\}} \mathit{reach}(z, S)\ .$$

So far we have considered only stores. A pointer structure (n, S) is called *acyclic, injective* or a *forest* if the store of its reachable part $from(n, S)$ is acyclic, injective or a forest, respectively.

We are now in the position to formulate the strong preservation properties for updating acyclic structures and hence forests.

Lemma 7.2 Assume that p is acyclic and $u \neq \varepsilon$. Then

1. $from((p.k := p.u).k) = from(p.u)$.
2. $j \neq k \Rightarrow from((p.k := p.u).j) = from(p.j)$.

If, moreover, F is a reasonable abstraction function, then

3. $(p.k := p.u).k \sim p.u$.
4. $j \neq k \Rightarrow (p.k := p.u).j \sim p.j$.

Moreover, we have

Lemma 7.3 Let $r = (p.k := q)$ and set $n \stackrel{\text{def}}{=} ptr(r) = ptr(p)$ and $R \stackrel{\text{def}}{=} \overline{\{n\}} \bowtie sto(r) = \overline{\{n\}} \bowtie sto(q)$.

1. If $ptr(q) \notin \{n\} \cup \mathsf{dom}[R]$ and R is acyclic, then r is acyclic as well.
2. If R is acyclic and $n \notin \{ptr(q)\} \cup sto(q)(\overline{\{n\}})$, then r is acyclic as well.
3. r is injective iff $ptr(q) \notin \mathsf{ran}[R]$ and R is injective.

Finally, we give a stronger criterion for acyclicity of overwritten structures:

Lemma 7.4 Consider $r \stackrel{\text{def}}{=} (n, S).k := (m, T) = (n, (n \stackrel{k}{\mapsto} m) \cup R)$ where $R \stackrel{\text{def}}{=} \overline{\{n\}} \bowtie T$. Then r is acyclic iff R is acyclic and $(m, R) \not\vdash n$.

8 Pointer Implementation of Lists

8.1 Abstract Lists

The set $\mathcal{L}$ of *binary trees* with elements of $\mathcal{N}$ as nodes is defined inductively as the least set $\mathcal{X}$ with

$$\varepsilon \cup \mathcal{N} \times \mathcal{X} \subseteq \mathcal{X} ,$$

where ε denotes the empty list. A non-empty tree, i.e., an element of $\mathcal{N} \times \mathcal{L}$, will be denoted as a pair $\langle x, l \rangle$ with head $x \in \mathcal{N}$ and tail $l \in \mathcal{L}$.

8.2 The Abstraction Function For Acyclic Structures

Let now P denote the set of all pointer structures over the record scheme for lists, as discussed in Section 3. The abstraction function $list : \mathcal{P} \rightsquigarrow \mathcal{L}$ constructs the list reachable from a record in a store. For $n \in \mathcal{A}$ we set

$$list(p) \stackrel{\text{def}}{=} \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \langle p.head, list(p.tail) \rangle .$$

The recursion pattern is typical of an *unfold operation* or *anamorphism* (see Meijer et al. (1991), Bird (1996)). In the case where a cycle is reachable from n in L, this recursion is non-terminating. In a strict underlying semantics this means that the value of $list(n, L)$ is undefined, whereas in a non-strict setting the value of $list(n, L)$ is an infinite list corresponding to an unwinding of the subgraph reachable from n in L. Since we are working in a relational setting, the strict interpretation is relevant here. So from now on we shall assume that *list* is used only for acyclic pointer structures. Below we will present an abstraction function that copes with cyclic lists. We have

Lemma 8.1 The abstraction function *list* is reasonable.

The proof is a straightforward adaptation of that of Lemma 22 in Möller (1997).

Lemma 8.2 An acyclic list pointer structure (m, L) with $m \in \mathcal{A}$ is a forest.

Proof: In the proof of Lemma 13 of Möller (1997) it was shown that in every Kleene algebra one has

$$a \cdot b \leq 1 \;\Rightarrow\; a^* \cdot b^* \leq a^* + b^* \ .$$

Hence for a map we obtain downward local linearity of R^*:

$$R^* \cdot (R^*)^\smile \subseteq R^* \cup (R^\smile)^* \qquad (*) \ .$$

Now assume $L_{tail}(k_1) = L_{tail}(k_2) = n$ for $k_1, k_2 \in reach(m, L)$ with $k_1 \neq k_2$. Then by $(*)$ w.l.o.g. $k_1 \ L_{tail}^+ \ k_2$. Since L_{tail} is a map it follows that $k_1 \ L_{tail} \ n \ L_{tail}^* \ k_2 \ L_{tail} \ n$, i.e., $n \ L_{tail}^+ \ n$, a contradiction to acyclicity of (m, L). ■

8.3 Concatenation

We now calculate pointer implementations of a number of sample operations. First we treat the operation $cat : \mathcal{L} \times \mathcal{L} \rightsquigarrow \mathcal{L}$ that concatenates two lists. It is recursively defined by

$$\begin{aligned} cat(\varepsilon, r) &= r \ , \\ cat(\langle x, l\rangle, r) &= \langle x, cat(l, r)\rangle \ . \end{aligned}$$

Using our general scheme from Section 5 we specify a general pointer implementation *pcat* by requiring *pcat* $POI(ALL)$ *cat* and want to find an explicit version of *pcat*. However, we will develop this only for the case of arguments without sharing.

So assume that $p = (m, n, L)$ with list entries $m, n \in \mathcal{A}$ is acyclic and that $\neg$ *sharing*(p) holds. Then the specification of *pcat* unfolds into

$$list(pcat(m, n, L)) = cat(list(m, L), list(n, L)) \ .$$

For the case that $m = \diamond$ we have $list(m, L) = \varepsilon$ and hence $list(pcat(m, n, L)) = list(n, L)$, so that we may choose

$$pcat(\diamond, n, L) = (n, L) \ .$$

For $m \neq \diamond$ we calculate, with $p = (m, L)$ and $q = (n, L)$,

$$
\begin{aligned}
& cat(list(p), list(q)) \\
= \quad & \{\![\text{ unfold definition of } list \ ]\!\} \\
& cat(\langle p.head, list(p.tail)\rangle, list(q)) \\
= \quad & \{\![\text{ unfold definition of } cat \ ]\!\} \\
& \langle p.head, cat(list(p.tail), list(q))\rangle \\
= \quad & \{\![\text{ fold with specification of } pcat, \text{ i.e., choose an arbitrary} \\
& \quad q' \in pcat(L_{tail}(m), n, L) \ ]\!\} \\
& \langle p.head, list(q')\rangle \\
= \quad & \{\![\text{ setting } r \stackrel{\text{def}}{=} p.tail := q' \text{ and using Lemma 6.2.5 }]\!\} \\
& \langle r.head, list(q')\rangle \\
= \quad & \{\![\text{ by Lemmas 6.3.2, 7.1.2 and 8.2, assuming additionally } pcat \subseteq pres \ ]\!\} \\
& \langle r.head, list(r.tail)\rangle \\
= \quad & \{\![\text{ fold with definition of } list \ ]\!\} \\
& list(r) \ .
\end{aligned}
$$

For the correctness of the folding step we observe that the assumption $\neg$ *sharing* propagates to the recursive call by Lemma 6.2.9. The derivation has shown the need for introducing the additional invariant *pres*; it is established by the termination case and propagated to r by a straightforward calculation using (1).

Altogether, for the following recursion we have *pcat* $POI(pres)$ *cat*:

$$pcat(m, n, L) = \text{if } m = \diamond \text{ then } (n, L) \text{ else } p.tail := pcat(L_{tail}(m), n, L)) \ .$$

8.4 Reversal

Next we want to derive a function for reversing a list. As reversal function $rev : \mathcal{L} \rightsquigarrow \mathcal{L}$ on abstract lists we use an embedding into a tail-recursive function $rrev : \mathcal{L} \times \mathcal{L} \rightsquigarrow \mathcal{L}$:

$$
\begin{aligned}
rev(l) &\stackrel{\text{def}}{=} rrev(l, \varepsilon), \\
rrev(\varepsilon, t) &\stackrel{\text{def}}{=} t \ , \\
rrev(\langle m, l\rangle, t) &\stackrel{\text{def}}{=} rrev(l, \langle m, t\rangle) \ .
\end{aligned}
$$

The additional parameter t of *rrev* accumulates the intermediate results. Using our general scheme from Section 5 we specify a general pointer implementation *prev* by requiring *prev* $POI(ALL)$ *rev* and want to find an explicit version of *prev*. We do this by finding a pointer implementation of the auxiliary function *rrev*. However, we do not carry the accumulating substructure itself as a parameter, but just its head cell. Hence we specify, assuming $\neg$ $sharing(m, n, L)$:

$$list(prrev(m, n, L) \ = \ rrev(list(m, L), list(n, L)) \ .$$

An appropriate embedding is $prev(m, L) = prrev(m, \diamond, L)$, since $list(\diamond, L) = \varepsilon$. As before, we now perform a case analysis.

Case 1: $m = \diamond$. Then $list(m, L) = \varepsilon$, and hence $rev(list(m, L)) = \varepsilon$. Thus $list(prrev(m, n, L) = list(n, L)$, so that we may choose $prrev(m, n, L) = (n, L)$ in this case.

Case 2: For $m \neq \diamond$ we calculate, with $p = (m, L)$ and $q = (n, L)$,

$$
\begin{aligned}
& list(prrev(m, n, L)) \\
= \quad & \{\![\text{ unfold specification of } prrev \ ]\!\} \\
& rrev(list(p), list(q)) \\
= \quad & \{\![\text{ unfold definition of } list \ ]\!\} \\
& rrev(\langle p.head, list(p.tail)\rangle, list(q)) \\
= \quad & \{\![\text{ unfold definition of } rrev \ ]\!\} \\
& rrev(list(p.tail), \langle p.head, list(q)\rangle) \\
= \quad & \{\![\text{ setting } r \stackrel{\text{def}}{=} p.tail := q \text{ and using Lemma 6.2.4 and 6.2.5 }]\!\} \\
& rrev(list(p.tail), \langle r.head, list(r.tail)\rangle) \\
= \quad & \{\![\text{ fold with definition of } list \ ]\!\} \\
& rrev(list(p.tail), list(r)) \\
= \quad & \{\![\text{ definition of } p \ ]\!\} \\
& rrev(list(L_{tail}(m), L), list(r)) \\
= \quad & \{\![\text{ by acyclicity of } p \text{ and Lemmas 6.3.1 and 7.1.2 }]\!\} \\
& rrev(list(L_{tail}(m), sto(r)), list(r)) \\
= \quad & \{\![\text{ fold with specification of } prrev \ ]\!\} \\
& list(prrev(L_{tail}(m), r)) \ .
\end{aligned}
$$

Hence we may choose $prrev(m, n, L) = prrev(L_{tail}(m), r)$ in this case. Altogether, we have obtained

$$
\begin{aligned}
& prev(m, L) = prrev(m, \diamond, L) \ , \\
& prrev(m, n, L) = \mathsf{if}\ m = \diamond\, \mathsf{then}\ (n, L) \\
& \qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{let}\ k = L_{tail}(m) \\
& \qquad\qquad\qquad\qquad\qquad\quad r = p.tail := q \\
& \qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ prrev(k, r) \quad .
\end{aligned}
$$

In this case no strengthening of the invariant ALL was necessary. Again, we have to check the validity of the assertion for the recursive call. Assume that $m \neq \diamond \wedge \neg\, sharing(m, n, L)$ hold, and set $k \stackrel{\text{def}}{=} L_{tail}(m)$ and $M \stackrel{\text{def}}{=} (m \stackrel{tail}{\mapsto} n) \mid L$. Then

$$reach(m, M) = \{m\} \cup reach(n, M) = \{m\} \cup reach(n, L)$$

by Lemma 6.1 and $\neg$ $sharing(m, n, L)$. Moreover,

$$reach(k, M) = reach(k, L) ,$$

again by Lemma 6.1. Now $\neg$ $sharing(k, m, M)$ is immediate using elementary set theory, acyclicity of (m, L), Lemma 7.1.2 and Lemma 6.2.9.

8.5 The Abstraction Function For Circular Lists

We now treat the case of circular lists. We now say that a pointer structure (m, L) represents the list which is obtained by following the links until an already visited record is reached. The corresponding abstraction function is $clist : P \rightsquigarrow \mathcal{L}$. For $m \in \mathcal{A}$ with $m = \diamond$ or cyclic (m, L) we set

$$clist(p) \stackrel{\text{def}}{=} \text{if } ptr(p) = \diamond \text{ then } \varepsilon \text{ else } \langle p.head, clis(p.tail, \{ptr(p)\})\rangle$$

where the auxiliary function $clis : \mathcal{P} \times \wp(\mathcal{A}) \rightsquigarrow \mathcal{L}$ is given by

$$clis(p, V) \stackrel{\text{def}}{=} \text{if } ptr(p) \in V \text{ then } \varepsilon \text{ else } \langle p.head, clis(p.tail, V \cup \{ptr(p)\})\rangle$$

Termination is now forced by the additional argument V of *clis* which recordmembers the set of already visited records. Again we have an anamorphic recursion pattern. First we show the following reasonableness properties for *clis*:

Lemma 8.3 1. For all $V \subseteq A$ the residual function $clis(\_, \_, V)$ is reasonable.
2. $ptr(p) = ptr(q) \wedge \overline{V} \bowtie p = \overline{V} \bowtie q \Rightarrow clis(p, V) = clis(q, V)$.

Proof: 1. In Möller 1997 it was shown that abstraction function F is reasonable if for all s, S, T we have $reach(s, S) \bowtie S = reach(s, S) \bowtie T \Rightarrow F(s, S) = F(s, T)$. We show the premise by fixpoint induction on the recursive definition of *clis* and the continuous predicate

$$PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, n, S, T, V : reach(n, S) \bowtie S = reach(n, S) \bowtie T \Rightarrow h(n, S, V) = h(n, T, V) .$$

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. The functional τ associated with the recursive definition of *clis* is

$$\tau(h)(q, V) = \text{if } ptr(q) \in V \text{ then } \varepsilon \text{ else } \langle q.head, clis(q.tail, V \cup \{ptr(q)\})\rangle .$$

First, $n \in reach(n, S)$ and $reach(n, S) \bowtie S = reach(n, S) \bowtie T$ imply

$$\begin{array}{l} \{n\} \bowtie S = \{n\} \bowtie T \wedge \\ reach(l, S) \bowtie S = reach(l, S) \bowtie T , \end{array} \qquad (*)$$

where $l = S_{tail}(n) = T_{tail}(n)$. We calculate

$$
\begin{array}{ll}
 & \tau(h)(n,S,V) \\
= & \quad \{\![\text{ definition }]\!\} \\
 & \text{if } n \in V \text{ then } \varepsilon \text{ else } \langle (n,S).head, clis((n,S).tail, V \cup \{n\})\rangle \\
= & \quad \{\![\text{ by } (*) \text{ and } PP(h) \ ]\!\} \\
 & \text{if } n \in V \text{ then } \varepsilon \text{ else } \langle (n,T).head, clis((n,T).tail, V \cup \{n\})\rangle \\
= & \quad \{\![\text{ definition }]\!\} \\
 & \tau(h)(n,T,V) \ .
\end{array}
$$

2. We use again fixpoint induction with the continuous predicate

$$
\begin{aligned}
PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, p_1, p_2, V : ptr(p_1) = ptr(p_2) \wedge \overline{V} \bowtie p_1 = \overline{V} \bowtie p_2 \Rightarrow \\
h(p_1, V) \ = \ h(p_2, V) \ .
\end{aligned}
$$

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. For the induction step we assume the premise of PP and set $m \stackrel{\text{def}}{=} ptr(p_1) = ptr(p_2)$. Then $reach(p_i)\backslash V = \{m\}\backslash V \cup reach(p_i.tail)\backslash V$.
Case 1: $m \in V$. By definition $\tau(h)(p_1, V) = \varepsilon = \tau(h)(p_2, V)$.
Case 2: $m \notin V$. Then by assumption $\{m\} \bowtie p_1 = \{m\} \bowtie p_2$ and hence $p_1.head = p_2.head$ and $ptr(p_1.tail) = ptr(p_2.tail)$. Moreover, since $V \subseteq V \cup \{m\}$, we have $\overline{V \cup \{m\}} \bowtie p_1.tail = \overline{V \cup \{m\}} \bowtie p_2.tail$. So the induction hypothesis is satisfied and we have

$$
\begin{aligned}
\tau(h)(p_1, V) = \langle p_1.head, clis(p_1.tail, V \cup \{m\})\rangle = \\
\langle p_2.head, clis(p_2.tail, V \cup \{m\})\rangle \tau(h)(p_2, V) \ .
\end{aligned}
$$

■

One can even show the stronger property

$$
ptr(p) = ptr(q) \ \wedge \ (reach(p)\backslash V) \bowtie p = (reach(q)\backslash V) \bowtie q \ \Rightarrow \ clis(p, V) = clis(q, V) \ .
$$

However, its premise is much harder to check than the one of 2. above, so that it is less useful. From 1. it is immediate that

Lemma 8.4 The abstraction function *clist* is reasonable.

The function *clis* also shows an important localization property:

Lemma 8.5 $p \not\Vdash W \ \Rightarrow \ clis(p, V \cup W) \ = \ clis(p, V)$.

Proof: We use again fixpoint induction with the continuous predicate

$$
PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, p, V, W : p \not\Vdash W \ = \ \emptyset \ \Rightarrow \ h(p, V \cup W) \ = \ h(p, V) \ .
$$

By Lemma 6.2.8 the premise of PP implies

$$
ptr(p) \notin W \ \wedge \ p.tail \not\Vdash W \ . \qquad (*)
$$

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. With τ as in the previous proof we calculate

$$
\begin{array}{ll}
 & \tau(h)(p, V \cup W) \\
= & \quad \{\![\text{ definition }]\!\} \\
 & \text{if } ptr(p) \in V \cup W \text{ then } \varepsilon \text{ else } \langle p.head,\, clis(p.tail, V \cup W \cup \{ptr(p)\})\rangle \\
= & \quad \{\![\text{ by } (*) \text{ and } PP(h) \,]\!\} \\
 & \text{if } ptr(p) \in V \text{ then } \varepsilon \text{ else } \langle p.head,\, clis(p.tail, V \cup \{ptr(p)\})\rangle \\
= & \quad \{\![\text{ definition }]\!\} \\
 & \tau(h)(p, V) \ .
\end{array}
$$

■

Finally we note that the representation of singleton lists is almost unique:

Lemma 8.6 $clist(q) = \langle x, \varepsilon\rangle \ \Leftrightarrow \ ptr(q) = ptr(q.tail) \wedge q.head = x.$

8.6 Concatenation of Circular Lists

Again we require *pcat* $POI(ALL)$ *cat*, but this time w.r.t. *clist*, and want to find an explicit version of *pcat* for the case of arguments without sharing.

So assume that $p = (m, L)$ and $q = (n, L)$ both represent circular lists and that $\neg\ sharing(m, n, L)$ holds. In particular then $m \neq n$. The specification of *pcat* now unfolds into

$$clist(pcat(m, n, L)) = cat(clist(p), clist(q)) \ .$$

For the case that $m = \diamond$, we have $clist(p) = \varepsilon$ and hence $clist(pcat(m, n, L)) = clist(n, L)$, so that we may choose

$$pcat(\diamond, n, L) = (n, L) \ .$$

For $m \neq \diamond$ we have to consider *clis*. We introduce an auxiliary function *pca*, also with additional parameter, that mirrors the reduction of *clist* to *clis*. It is specified by

$$clis(pca(m, n, L, V), V) = cat(clis(m, L, V), clis(n, L, V)) \ .$$

Case 1: $m \in V$. Then $clis(m, L, V) = \varepsilon$ and hence $clis(pca(m, n, L, V), V) = clis(n, L, V)$, so that we may choose $pca(m, n, L, V) = (n, L)$ in this case.
Case 2: $m \notin V$. Set $p = (m, L)$ and $q = (n, L)$. We calculate

$$
\begin{array}{ll}
 & cat(clis(p, V), clis(q, V)) \\
= & \quad \{\![\text{ unfold definition of } clis \,]\!\} \\
 & cat(\langle p.head, clis(p.tail, V \cup \{m\})\rangle, clis(q, V)) \\
= & \quad \{\![\text{ unfold definition of } cat \,]\!\} \\
 & \langle p.head, cat(clis(p.tail, V \cup \{m\}), clis(q, V))\rangle \\
= & \quad \{\![\text{ by Lemma 8.5 }]\!\} \\
 & \langle p.head, cat(clis(p.tail, V \cup \{m\}), clis(q, V \cup \{m\}))\rangle
\end{array}
$$

$=$ {[fold with specification of pca setting $l \stackrel{\text{def}}{=} ptr(p.tail)$]}

$\langle p.head, clis(pca(l, n, L, V \cup \{m\}), V \cup \{m\})\rangle$

$=$ {[fold with specification of $pcat$, i.e., choose an arbitrary $q' \in pca(l, n, L, V \cup \{m\})$]}

$\langle p.head, clis(q')\rangle$

$=$ {[setting $r \stackrel{\text{def}}{=} p.tail := q'$ and using Lemma 6.2.5]}

$\langle r.head, clis(q', V \cup \{m\}))\rangle$

$=$ {[fold with definition of $clis$ using $ptr(r) = m$ by Lemma 6.2.1]}

$clis(r, V)$.

Again the assumption $\neg$ *sharing* propagates to the recursive call by Lemma 6.2.9. Altogether we have obtained

$$\begin{aligned} pcat(m, n, L) &= \mathsf{if}\ m = \diamond\ \mathsf{then}\ (n, L) \\ &\qquad \mathsf{else}\ p.tail := pca(L_{tail}(m), n, L, \{m\}) \\ pca(m, n, L, V) &= \mathsf{if}\ m \in V\ \mathsf{then}\ (n, L) \\ &\qquad \mathsf{else}\ p.tail := pca(L_{tail}(m), n, L, V \cup \{m\}) \end{aligned}$$

Note the close correspondence between the derivations in the acyclic and in the circular case.

8.7 Reversal of Cyclic Lists

Assume now the specification

$$clist(prev(p)) = rev(clist(p)) \ .$$

We have

$rev(clist(p))$

$=$ {[definitions of $clist$ and rev]}

$\mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ rrev(clis(p.tail, \{ptr(p)\}), \langle p.head, \varepsilon\rangle)$.

$=$ {[definition of $clist$ and Lemma 8.6]}

$\mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ clist(p)\ \mathsf{else}\ rrev(clis(p.tail, \{ptr(p)\}), clist(p.tail := p))$.

This calls for the introduction of an auxiliary function $prrev$ specified by

$$clis(prrev(m, n, L, V), V) = rrev(clis(m, L, V), clis(n, L, V))$$

provided $\neg$ *sharing*(m, n, L, V). Set $p \stackrel{\text{def}}{=} (m, L)$ and $q \stackrel{\text{def}}{=} (n, L)$.

Case 1: $m \in V$. We have $rrev(clis(p, V), clis(q, V)) = clis(q, V)$, so that we may choose $prrev(m, n, L, V) = q$ in this case.

Case 2: $m \notin V$. We calculate

$$
\begin{aligned}
& rrev(clis(p, V), clis(q, V)) \\
= \quad & \{\![\text{ unfold definition of } clis \ ]\!\} \\
& rrev(\langle p.head, clis(p.tail, V \cup \{m\})\rangle, clis(q, V)) \\
= \quad & \{\![\text{ unfold definition of } rrev \ ]\!\} \\
& rrev(clis(p.tail, V \cup \{m\}), \langle p.head, clis(q, V)\rangle) \\
= \quad & \{\![\text{ by Lemma 8.5 }]\!\} \\
& rrev(clis(p.tail, V \cup \{m\}), \langle p.head, clis(q, V \cup \{m\})\rangle) \\
= \quad & \{\![\text{ setting } r \stackrel{\text{def}}{=} p.tail := q \text{ and using Lemma 6.2.4 }]\!\} \\
& rrev(clis(p.tail, V \cup \{m\}), \langle r.head, clis(r.tail, V \cup \{m\})\rangle) \\
= \quad & \{\![\text{ fold with definition of } clis \ ]\!\} \\
& rrev(clis(p.tail, V \cup \{m\}), clis(r, V)) \\
= \quad & \{\![\text{ by Lemma 8.5 using } \neg\, sharing(m, n, L, V) \ ]\!\} \\
& rrev(clis(p.tail, V \cup \{m\}), clis(r, V \cup \{m\})) \\
= \quad & \{\![\text{ definition of } p \ ]\!\} \\
& rrev(clis(L_{tail}(m), L, V \cup \{m\}), clis(r, V \cup \{m\})) \\
= \quad & \{\![\text{ by Lemma 8.3.2 }]\!\} \\
& rrev(clis(L_{tail}(m), sto(r), V \cup \{m\}), clis(r, V \cup \{m\})) \\
= \quad & \{\![\text{ fold with specification of } prrev \ ]\!\} \\
& clis(prrev(L_{tail}(m), r, V \cup \{m\})) \ .
\end{aligned}
$$

Hence we may choose

$$prrev(m, n, L, V) \ = \ prrev(L_{tail}(m), r, V \cup \{m\})$$

in this case. Altogether, we obtain, using Lemma 8.6 and once more Lemma 8.3.2,

$$
\begin{aligned}
prev(m, L) = \ & \mathsf{if}\ m = \Diamond\, \mathsf{then}\ (m, L) \\
& \qquad \mathsf{else}\ prrev(L_{tail}(m), m, (m \stackrel{tail}{\mapsto} m) \mid L, \{m\}) \\
prrev(m, n, L, V) = \ & \mathsf{if}\ m \in V\ \mathsf{then}\ (n, L) \\
& \qquad \mathsf{else}\ \mathsf{let}\ k = L_{tail}(m) \\
& \qquad\qquad r = p.tail := q \\
& \qquad\quad \mathsf{in}\ \ prrev(k, r, V \cup \{m\}) \quad .
\end{aligned}
$$

Again, the derivation was very similar to the one for the acyclic case.

9 Conclusion and Outlook

The relational calculus has proved to be a very useful tool for modeling and analyzing pointer structures. The chosen abstraction seems adequate, as the fairly

concise derivations in the examples show. It is encouraging that to a large extent the treatment is independent of the particular data structures involved. The extension to properly cyclic structures has proved to be relatively simple and did not need additional concepts.

It remains to integrate the approach with the general theory of unfold operations or anamorphisms. The proof of Lemmas 8.1 and 8.3 leads us to conjecture that every anamorphic abstraction function is reasonable. Moreover, the similarity of the derivations in the acyclic and cyclic cases suggest that there must be a unifying approach to these. This will be left to subsequent papers.

Acknowledgements This research was partially sponsored by Esprit Working Group 8533 *NADA — New Hardware Design Methods.*

References

1. U. Berger, W. Meixner, B. Möller: Calculating a garbage collector. In: M. Broy, M. Wirsing (eds.): Methods of programming. Lecture Notes in Computer Science **544**. Berlin: Springer 1991, 137–192
2. R. Bird: Functional algorithm design. Science of Computer Programming **26**, 15–31 (1996)
3. R.S. Bird, O. de Moor: Algebra of programming. Prentice-Hall 1996
4. C.A.R. Hoare: Proofs of correctness of data representations. Acta Informatica **1**, 271–281 (1972)
5. E.Meijer, M.Fokkinga, R. Paterson: Functional programming with bananas, lenses, envelopes and barbed wire. In: J. Hughes (ed.): Functional programming and computer architecture. Lecture Notes in Computer Science **523**. Berlin: Springer 1991, 124–144
6. B. Möller: Formal derivation of pointer algorithms. In: M. Broy (Hrsg.): Informatik und Mathematik. Berlin: Springer1991, 419–440
7. B. Möller: Development of graph and pointer algorithms. In: B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Lecture Notes in Computer Science **755**. Berlin: Springer 1993, 123–160
8. B. Möller: Towards pointer algebra. Science of Computer Programming **21**, 57–90 (1993)
9. B. Möller: Calculating with pointer structures. In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997. Chapman&Hall 1997 (to appear)
10. G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993